

本周主要是实现 SDL 的索引与查找，按照论文中的描述构造了四层树形结构，如图 1 所示。第一层是对不同形状的符号的记录，图中共有八个。第二层则记录了每个符号在序列中出现的起始位置，每个节点用一个数组表示，假设时间序列最大长度为  $np$ ，则每个数组的长度为  $np$ 。第三层记录的是每个符号连续出现的次数，节点数组的长度与符号的起始位置是相关的，假设起始位置为  $k$ ，则连续出现的次数最多不超过  $np-k$ ，因此数组长度定义为  $np-k$ 。第四层记录的是时间序列，通常用序列的 id 指代。

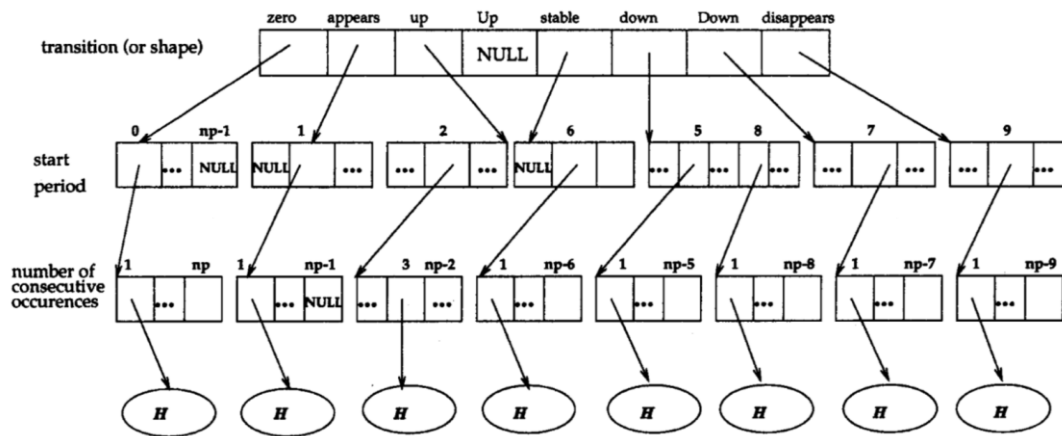


Figure 2: An index structure for *SDL* queries.

## 1、数据表示

从第三层开始，分别用三个类来表示。

第三层为 Occurrence

```
class Occurrence {
    List<String> series;
    int consecutiveNum;
    String pointer;

    public Occurrence() {
        consecutiveNum = 0;
        series = new ArrayList<>();
    }
}
```

第二层为

```
class StartPeriod {
    Occurrence[] occur;

    public StartPeriod(int m, int s) {
        occur = new Occurrence[m-s];
        for (int i = 0; i < m-s; i++) {
            occur[i] = new Occurrence();
        }
    }
}
```

第一层为

```
class Transition {
    String alphabet;
    StartPeriod[] period;
    String pointer;

    public Transition(int m) {
        period = new StartPeriod[m];
        for (int i = 0; i < m; i++) {
            period[i] = new StartPeriod(m, i);
        }
    }
}
```

## 2、插入操作

给定一个时间序列，如何将序列放入上述索引结构中？插入操作即使完成此功能。简单的做法是，按照索引的层次级别，按顺序遍历时间序列，并且根据符号将序列 id 记录到不同的数组中，具体代码如下。

```
public void insert(String[] series, String seriesId) {
    Transition[] tran = node.getTransitions();

    // Read time series in order
    for (int idx = 0; idx < series.length; ) {
        int pos = 0;    // start period
        String s = series[idx];
        StartPeriod sp = null;
        if (s.equals("zero")) {
            sp = tran[0].period[pos];

        } else if (s.equals("appears")) {
            sp = tran[1].period[pos];
        } else if (s.equals("up")) {
            sp = tran[2].period[pos];

        } else if (s.equals("Up")) {
            sp = tran[3].period[pos];

        } else if (s.equals("stable")) {
            sp = tran[4].period[pos];

        } else if (s.equals("down")) {
            sp = tran[5].period[pos];

        } else if (s.equals("Down")) {
            sp = tran[6].period[pos];
        }
    }
}
```

```

    } else if (s.equals("disappears")) {
        sp = tran[7].period[pos];
    }

    // scan symbols to get consecutive number
    int cnum = 0;
    for (int i = idx; i < series.length; i++) {
        if (series[i].equals(s)) {
            cnum ++;
            idx ++;
            continue;
        } else {
            break;
        }
    }

    sp.occure[cnum].series.add(seriesId);
}
}

```

经过测试，能够正确建立空气污染数据的索引结构。

### 3、查询操作

给定一个形状，获得最相似的时间序列，需要从上述索引中寻找。从索引结构中可以看出，任意给定一个形状，如(shape spike (concat Up up down Down))，需要按照顺序从索引中逐个查找。也就是说，上例中，首先要找到所有以 Up 开头仅出现一次的时间序列，再找所有以 up 在第二位仅出现一次，接下来找 down 在第三位的仅出现一次，最后找 Down 在第四位且仅出现一次的时间序列，最后所有的时间序列集合交集就是最终结果。按照这样的查询未免太过繁琐，而且查询效率较低。四个集合需要执行集合的交运算，时间复杂度为  $O(n^2)$ ，空间上也较大。

为此，重新阅读了与时间序列相关的论文，SAX 以符号表示时间序列，能够减少维度并且能计算距离（按时间序列的包围盒计算，即 lower bounding 方法）。然而无法满足我们查询形状的需要。论文中提出了 SAX 的索引方法，采取了一种近似索引方法 Vector Approximation。这种方法首先并不对原数据进行搜索，而是先搜索一个从原数据抽取出来的近似数据，减少第一次扫描的数据量。得到这些近似数据后，再次搜索原数据获得结果。

但是抽取近似数据也比较麻烦，考虑到 SDL 中数据维度并不高，可以考虑使用 R 树构建索引。第一层仍然是固定的符号，然后分别按符号出现的顺序逐层划分空间。R 树索引的缺陷在于维度不能太高，所以查询的时间序列长度不能太长。对于我们的应用来说，用户查询的形状不会太长（限制在 8），否则超出了形状难以理解。